Debugging Logical Errors

Debug Logical Errors By Checking Inputs and Outputs

When your code generates a logical error, the best thing to do is **compare the expected output to the actual output**.

1. Copy the function call from the `assert` that is failing into the interpreter. Compare the actual output to the expected output.
    - `assert` functions work by throwing an assertion error if the expression inside them is false
2. If the **expected** output seems incorrect, re-read the problem prompt.
3. If you're not sure why the actual output is produced, use a **debugging process** to investigate.

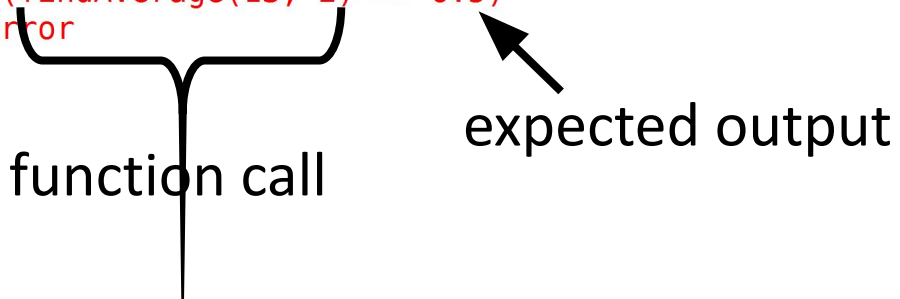
If you've written the test set yourself, you should also take a moment to make sure the test itself is not incorrect.

example.py

```
1  def findAverage(total, n):
2      if n <= 0:
3          return "Cannot compute the average"
4      return total // n
5
6  def testFindAverage():
7      print("Testing findAverage()...", end="")
8      assert(findAverage(20, 4) == 5)
9      assert(findAverage(13, 2) == 6.5)
10     assert(findAverage(10, 0) == "Cannot compute the average")
11     print("... done!")
12
13 testFindAverage()
```

```
Running script: "C:\Users\river\Downloads\example.py"
Testing findAverage()...Traceback (most recent call last):
  File "C:\Users\river\Downloads\example.py", line 13, in
<module>
    testFindAverage()
  File "C:\Users\river\Downloads\example.py", line 9, in t
estFindAverage
    assert(findAverage(13, 2) == 6.5)
AssertionError

>>>
```



Understanding the Prompt

When something goes wrong with your code, before rushing to change the code itself, you should make sure you understand **conceptually** what your code does.

First- make sure you're solving the right problem! Re-read the problem prompt to check that you're doing the right task.

It can help to analyze the **test cases** to make sure you understand why each input results in each output.

Making Hypotheses

If something looks wrong in the printed results, make a hypothesis about what the problem is and adjust your code accordingly. Then run the code again and see if the values change. Repeat this as much as necessary until your code works as expected.

An important part of this process is that you have to be intentional about the changes you make. Don't just change parts of the code haphazardly!

Sidebar: Clean Up Top-Level Testing

Some students like to test their code by adding print statements and function calls at the top level of the code (not inside a function).

This is fine, but if you do this, **remove the top-level code** before you submit on Gradescope because it might confuse the autograder!!

Alternative approach: do testing in the shell/interpreter! After you run your file, all of your functions are available there to be tested.

Ways to Debug

There are many approaches you can take towards debugging code effectively, including:

- **Rubber Duck Debugging:** talking through your code
- **Printing and Experimenting:** visualizing what's in your code
- **Thorough Tracing:** checking each part of the code line-by-line

Rubber Duck Debugging

Explain what your code is supposed to do and what is going wrong out loud to an inanimate object, like a rubber duck.

In the process of explaining your code out loud, you may find that a piece of your code does not match your intentions or a missing step.
This works more often than you might think!

Print and Experiment

Or try **printing and experimenting** to determine where in your code the problem is.

Add print statements strategically around where you think the error occurs. Run the code again and check whether the printed values match what you think they should be at that stage in the code.

You can also make your print statements more informative - e.g. add a brief string to indicate what's being printed and where.

```
# f is some function
def foo():
    x = f(1)
    y = f(2)

    print("Before if x=", x, "y=", y)
    if x < 10:
        y += 100
        print("In if y=", y)
    elif y < 10:
        x += 100
        print("In elif x=", x)
    else:
        x += y
        print("In else x=", x)
    print(x, y)
    return x + y
```

Thorough Code Tracing

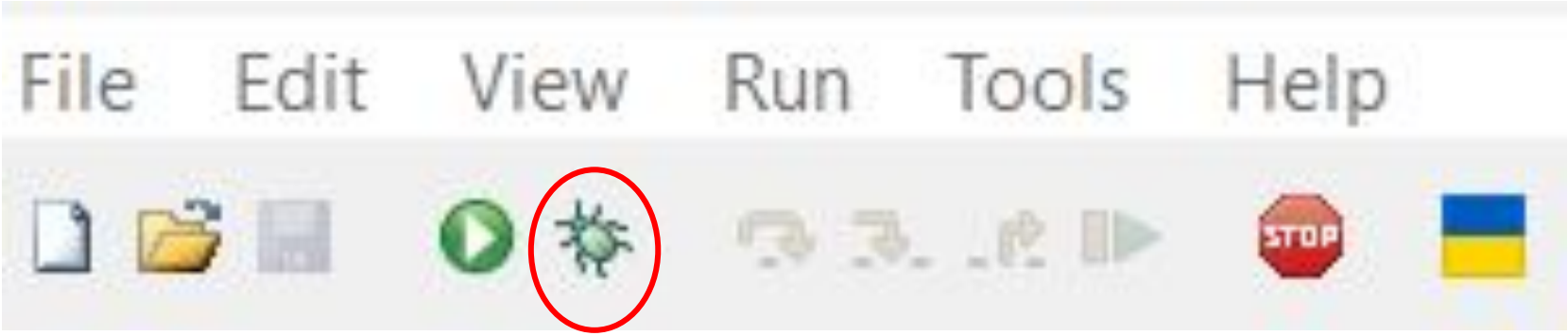
If you can't find the problem through printing and experimenting, you may have to resort to **code tracing** to determine what's going wrong.

Step through your code line by line and track **on paper** what values should be held in each of your variables at each step of the process.

Compare your traced values with what values your code should be producing. This might help you identify where the problem is occurring.

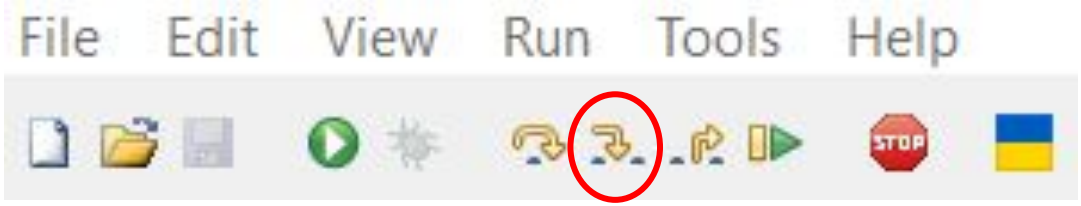
Tracing with Tools

Learning how to trace code by hand is a useful skill, but there are also **tools** that can help support you during debugging. One such tool is actually built right into Thonny! To access it, you just need to click the bug icon next to the green run arrow.



Using Thonny's Debugger

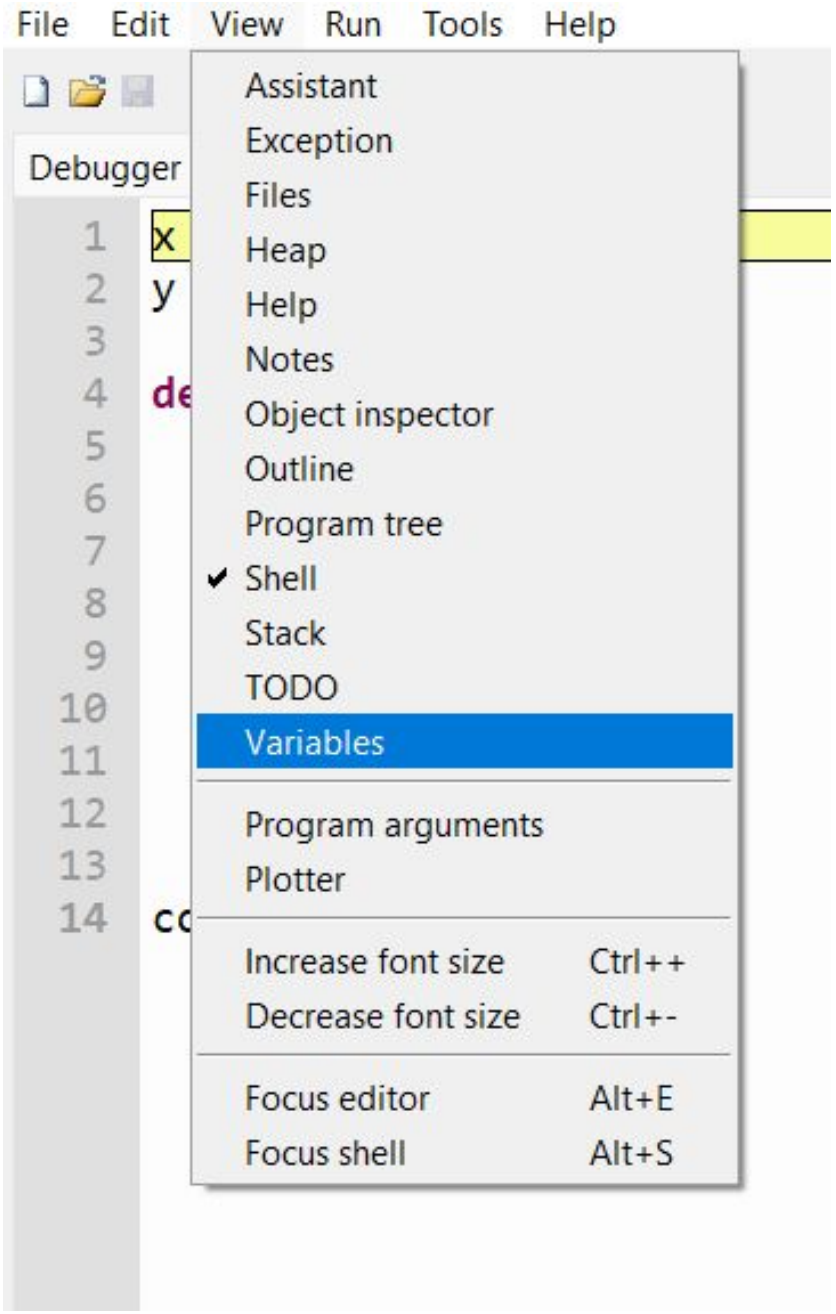
Once you've selected the bug icon, the menu will look like this



Select the "Step Into" Arrow (circled in red) to progress through the code bit by bit. It will assign variables their values and complete calculations one step at a time to show you exactly what is happening in your code.

Using Thonny's Debugger

If you would like to see the variables that you are currently working with in your code and what values they hold, simply select view -> variables



Additional Code Tracing Resources

If you would like to learn more about how to use Thonny's debugger, visit the bonus slides here.

If you would like to try a different resource you can also check out the website http://pythontutor.com/ .

Debugging is Hard

Finally, remember that debugging is hard! If you've spent more than **15 minutes stuck on an error**, more effort is not the solution. Get a friend or TA to help (or Piazza!), or take a break and come back to the problem later. A fresh mindset will make finding your bug much easier.